Memoria P1 - CN

Práctica Entregable: Diseño de Aplicaciones en la Nube

Roberto Tejero Martín

10/11/2025

Índice

1. Introducción y Objetivos

El objetivo de esta práctica es el diseño y despliegue de una aplicación API REST robusta y escalable utilizando servicios fundamentales de AWS (Amazon Web Services), tal como se especifica en los requisitos de la entrega.

El proyecto consiste en una API de gestión de Tareas (Tasks) que expone cinco endpoints, cubriendo todas las operaciones CRUD obligatorias:

- POST /task: Crear un nueva tarea.
- GET /tasks: Obtener todos las tareas.
- GET /task/{id}: Obtener una tarea mediante su ID.
- PUT /task/{id}: Actualizar una tarea.
- DELETE /task/{id}: Eliminar una tarea.

La api se ha desarrollado en Node.js siguiendo una arquitectura Clean Code. Esta base de código da soporte a dos infraestructuras de despliegue en AWS:

1. **Arquitectura Acoplada (Monolítica):** Un servidor API tradicional desplegado mediante contenedores en ECS Fargate, detrás de un balanceador de carga (NLB) y un API Gateway.
2. **Arquitectura Desacoplada (Serverless):** Una solución nativa de la nube donde cada endpoint es gestionado por una función AWS Lambda independiente, invocada también a través de API Gateway.

Ambas arquitecturas utilizan AWS API Gateway como fachada pública y AWS DynamoDB como base de datos gestionada, cumpliendo así los requisitos de desacoplamiento y cómputo en la nube. Todo el despliegue de recursos se realiza mediante plantillas de AWS CloudFormation (YAML).

Todo el contenido de esta práctica y los elementos a los que se hace referencia se encuentra disponible en el repositorio CN-P1.

2. Componentes Comunes

Antes de desplegar cualquiera de las arquitecturas, se deben desplegar dos recursos comunes que ambas utilizarán.

2.1. AWS ECR (Elastic Container Registry)

Ambas arquitecturas (acoplada y desacoplada) despliegan su código a partir de imágenes Docker. La plantilla `ecr.yml` crea un repositorio ECR para almacenar dichas imágenes.

- **Recurso Principal:** AWS::ECR::Repository.
- **Política de Ciclo de Vida:** Se ha configurado una política que expira automáticamente las imágenes antiguas, conservando solo las 2 más recientes para optimizar costes de almacenamiento.

2.2. AWS DynamoDB (Base de Datos)

La plantilla `db_dynamodb.yml` crea la tabla NoSQL que servirá como persistencia de datos para las tareas.

- **Recurso Principal:** AWS::DynamoDB::Table.
- **Esquema:** Se define una tabla simple con `task_id` como Clave de Partición (HASH Key) de tipo String (`AttributeType: S`).

3. Arquitecturas Propuestas

Para cumplir con los requisitos de la práctica, se han diseñado dos implementaciones de la misma API, correspondiendo a las arquitecturas recomendadas: una "versión acoplada" (ECS) y una "versión desacoplada" (Lambda). Ambas utilizan los componentes comunes (ECR y DynamoDB) y exponen la API a través de una API Gateway.

3.1. Arquitectura 1: Acoplada (ECS Fargate)

Este modelo representa un enfoque más tradicional, donde un servidor de aplicaciones monolítico (el servidor Node.js/Express que se ha desarrollado) se ejecuta de forma continua en un contenedor.

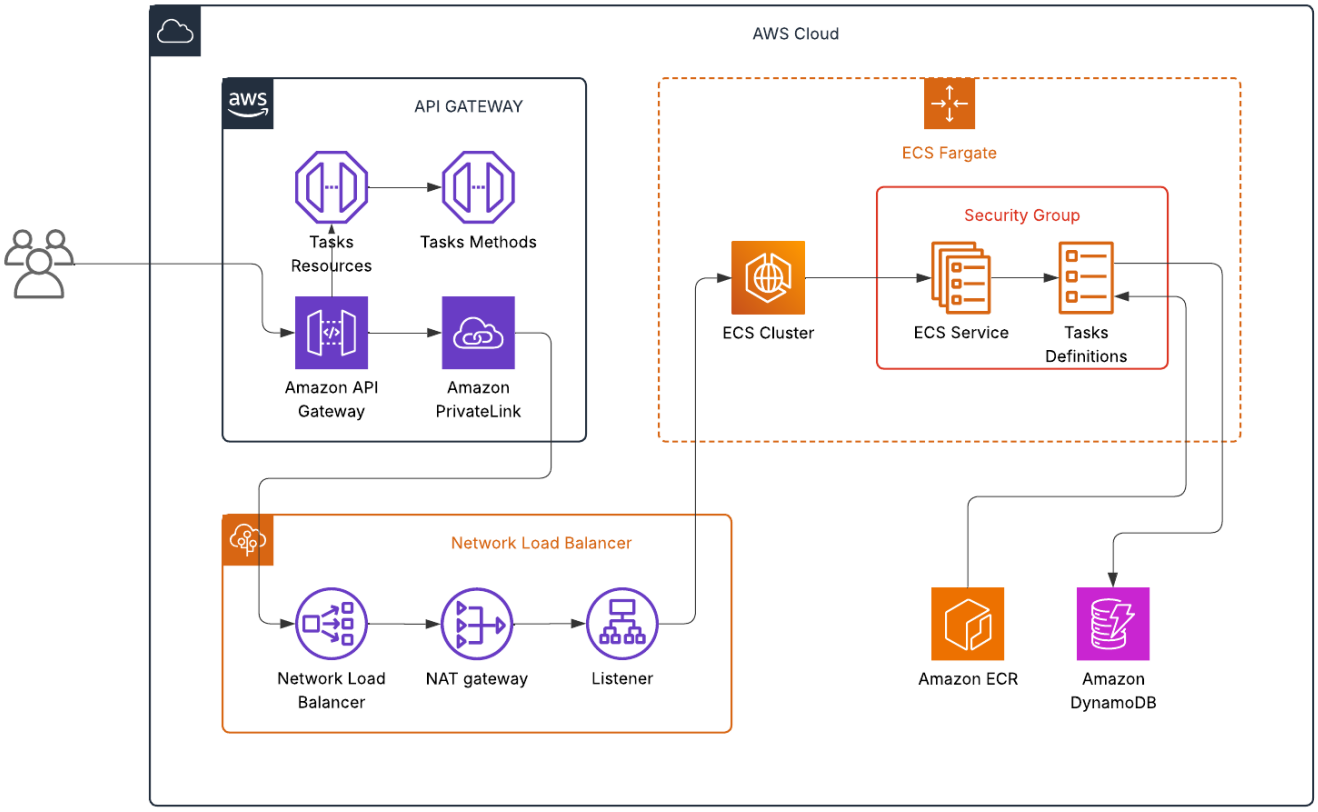

Figura 1: Diagrama de arquitectura acoplada.

Componentes Clave

1. **API Gateway (RestAPI):** Actúa como punto de entrada público (bastión), gestiona la autenticación (requiere API Key) y la documentación de la API.
2. **VPC Link:** Es el enlace privado entre el servicio público (API Gateway) y los recursos dentro de la VPC definida. Permite a API Gateway invocar al balanceador de carga interno (NLB) sin exponerlo a Internet.
3. **Network Load Balancer (NLB):** Se optó por utilizar un NLB (Network Load Balancer), debido a que ofrece un alto rendimiento para el tráfico TCP al operar en la capa de transporte del protocolo IP, además de tener un bajo coste. Se configuró como interno, lo que implica que solo gestionará el tráfico proveniente del VPC Link.
4. **ECS Fargate Service:** Es el orquestador (`AWS::ECS::Service`) que asegura que el número deseado de tareas (`DesiredCount: 1`) esté siempre en ejecución. Se utiliza el `LaunchType: FARGATE`, eliminando la necesidad de gestionar los servidores EC2 subyacentes.
5. **Task Definition:** Define la imagen Docker a usar (`ImageName`), los recursos (CPU/Memoria), los roles de IAM y las variables de entorno que el contenedor necesita para conectarse a DynamoDB.
6. **Integración:** API Gateway utiliza una integración de tipo `HTTP_PROXY`, reenviando las peticiones directamente al listener del NLB.

Flujo de Petición

1. El cliente realiza una petición a "https. . . /prod/{endpoint}" con su `x-api-key` en la cabecera.
2. El API Gateway valida la clave y enruta la petición a través del VPC Link.
3. El NLB interno recibe la petición en el puerto 8080 y la reenvía a una tarea registrada en el Target Group.
4. La tarea Fargate recibe la petición HTTP en su puerto 8080.
5. El servidor Express contenido en el Docker procesa la petición, opera con DynamoDB y resuelve la petición.
6. La respuesta viaja por el mismo camino (NLB $\rightarrow$ VPC Link $\rightarrow$ API GW) de vuelta al cliente.

3.2. Arquitectura 2: Desacoplada (Serverless Lambda)

Este modelo presenta una arquitectura nativa de la nube y que no requiere un servidor (serverless). Esto elimina la necesidad de gestionar servidores, ya que el cómputo se activa y se ejecuta solo en respuesta a una petición.

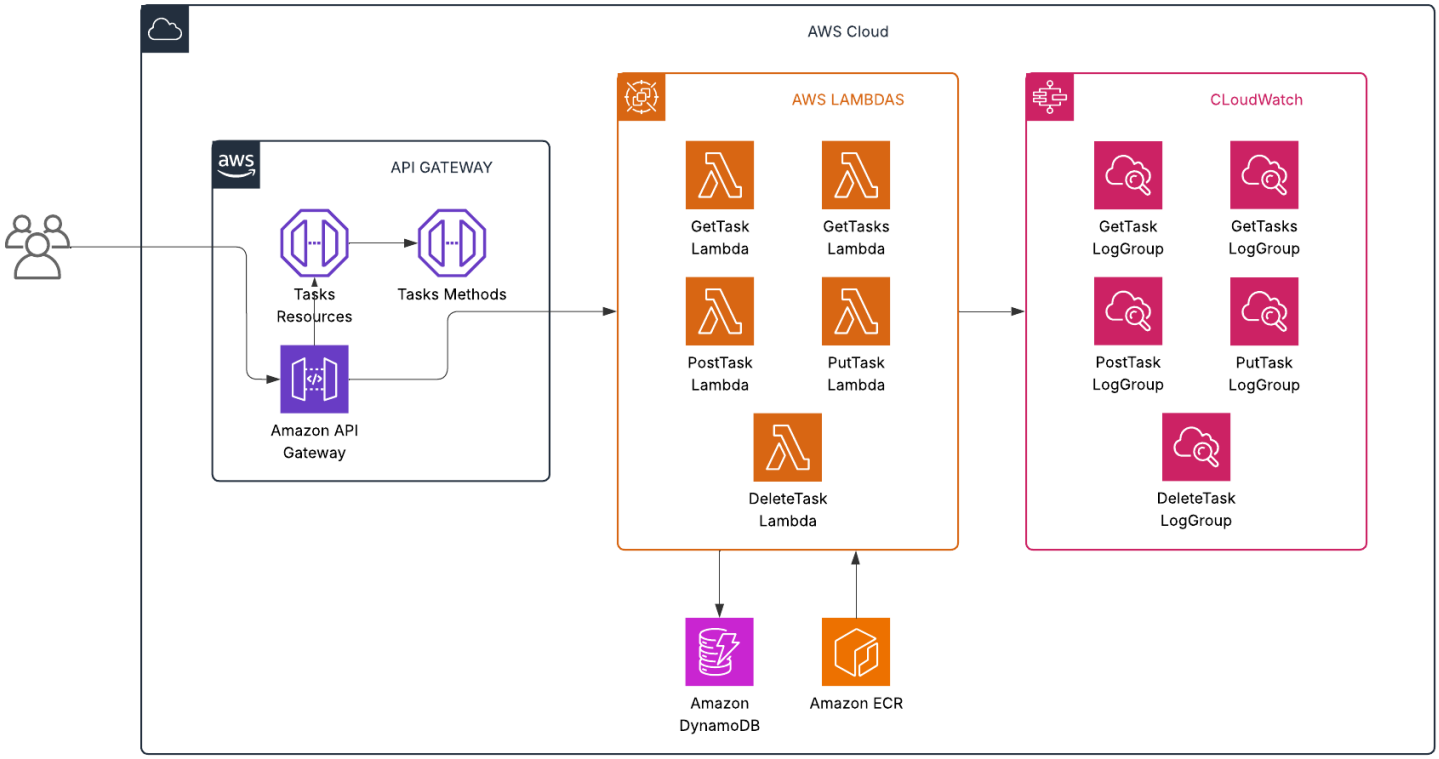

Figura 2: Diagrama de arquitectura desacoplada.

Componentes Clave

1. **API Gateway (RestAPI):** Similar al caso anterior, gestiona las rutas, la API Key y la documentación.
2. **Funciones AWS Lambda (x5):**
   a) **Cinco funciones Lambda distintas:** una para cada operación CRUD (CreateTaskLambda, GetAllTasksLambda, ...), cumpliendo el límite de 10 funciones (5 ya creadas + 5 creadas para esta arquitectura).

   b) **Despliegue desde Imagen:** Todas las funciones utilizan `PackageType: Image` y se despliegan desde la misma imagen ECR optimizando la gestión de dependencias.

   c) **Selección de Handler:** La selección de la operación se logra mediante la propiedad `ImageConfig.Command`. Esta propiedad sobrescribe el `CMD` del Dockerfile e indica a cada Lambda qué handler específico debe ejecutar dentro del código de la imagen.

3. **Integración:** La API Gateway utiliza la integración `AWS_PROXY`. En lugar de solo reenviar la petición HTTP, empaqueta toda la información (headers, body, path) en un evento JSON y lo pasa a la función Lambda.
4. **Log Groups:** Se crea un Log Group de CloudWatch para cada función y los permisos (`AWS::Lambda::Permission`) necesarios para que la API Gateway pueda invocarlas.

Flujo de peticiones

1. El cliente realiza una petición a "https.../prod/{endpoint}" con el JSON de la correspondiente y su `x-api-key` definida en la cabecera.
2. API Gateway valida la clave y, basándose en la ruta y el método, invoca directamente a la lambda correspondiente.
3. La función Lambda se despierta (si estaba en estado "fría") y ejecuta el handler especificado en `ImageConfig.Command`.
4. El código del handler parsea el body del evento, cumple con la petición y devuelve el resultado en una respuesta JSON (con `statusCode` y `body`).
5. API Gateway recibe este objeto y lo formatea como una respuesta HTTP estándar (ej. 201 Created) para el cliente.

4. Puesta en Marcha (Proceso de Despliegue)

Para lanzar la infraestructura completa, se debe seguir un orden específico de despliegue:

Nota: En los comandos que se muestren a continuación se deben reemplazar las variables señaladas con {{}} por su valor real.

Paso 1: Desplegar ECR

En primer lugar, se debe lanzar la plantilla `ecr.yml`. Esta plantilla despliega el repositorio de contenedores y requiere el parámetro `RepositoryName`, que tiene el valor por defecto de `tasks-app`. Es importante tener en cuenta este nombre, ya que se usará en los pasos de subida de la imagen.

Debido a que ambas arquitecturas requieren una imagen propia, es necesario desplegar un servicio ECR para cada una de las arquitecturas.

Paso 2: Desplegar de la Base de Datos

A continuación, se lanza la plantilla `db_dynamodb.yml`. Esta acción crea la tabla de DynamoDB que dará persistencia a la aplicación, utilizando el parámetro `TableName` que por defecto es `tasks`.

Este recurso es común a ambas arquitecturas y puede ser empleado simultáneamente por ambas.

Paso 3: Construir y Subir Imágenes Docker

Una vez creados el repositorio ECR y la base de datos, el siguiente paso es construir las imágenes Docker para cada arquitectura y subirlas al repositorio.

Subida de imagen a ECR:

```
AWS --password-stdin {{id_cuenta}}.dkr.ecr.{{region}}.amazonaws.com

docker build --platform linux/amd64 -t {{nombre_docker}} -f \
{{ruta_dockerfile}} . --provenance=false
```

```
docker tag tasks-acoplada:latest \
{{id_cuenta}}.dkr.ecr.us-east-1.amazonaws.com/{{nombre_ecr}}:latest

docker push \
{{id_cuenta}}.dkr.ecr.us-east-1.amazonaws.com/{{nombre_ecr}}:latest
```

Paso 4: Desplegar la Arquitectura

Opción A: Arquitectura Acoplada Lanzar la plantilla `acoplada.yml`. Se deben proporcionar los siguientes parámetros:

- **ImageName:** El nombre y tag de la imagen subida. (Default: `tasks-acoplada:latest`).
- **SubnetIds:** Seleccionar al menos 2 subnets de la VPC.
- **VpcId:** Seleccionar la VPC donde se desplegarán los recursos.
- **VpcCidr:** El bloque CIDR de la VPC seleccionada (ej. `10.0.0.0/16`).
- **DBDynamoName:** El nombre de la tabla de DynamoDB creada en el Paso 2. (Por defecto: `tasks`).

Opción B: Arquitectura Desacoplada Lanzar la plantilla `desacoplada.yml`. Se deben proporcionar los siguientes parámetros:

- **ImageName:** El nombre de la imagen subida. (Por defecto: `tasks-desacoplada:latest`).
- **DBDynamoName:** El nombre de la tabla de DynamoDB creada en el Paso 2. (Por defecto: `tasks`).

5. Seguridad de la Arquitectura

Se ha implementado una estrategia de seguridad en múltiples capas, protegiendo tanto el punto de entrada (API) como la infraestructura de cómputo y la base de datos.

5.1. Seguridad de Acceso (API Key)

- **Creación:** Ambas plantillas (`acoplada.yml` y `desacoplada.yml`) crean un recurso `AWS::ApiGateway::ApiKey` y lo asocian a un `AWS::ApiGateway::UsagePlan`.
- **Uso:** Todos los métodos CRUD de la API (excepto las peticiones `OPTIONS` para CORS) están configurados con la propiedad `ApiKeyRequired: true`. Esto obliga a que cualquier cliente que consuma la API deba incluir una cabecera `x-api-key` válida en su petición.
- **Obtención de la Clave:** CloudFormation genera un ID para la API Key (emitido en los Outputs como `LambdaAPIKeyId` o `ECSAPIKeyId`), pero no su valor secreto. Para obtener el valor real que se debe usar en la cabecera `x-api-key`, se debe ejecutar el siguiente comando de AWS CLI:

```
# Reemplazar <YOUR_API_KEY_ID> con el ID del Output de CloudFormation
aws apigateway get-api-key --api-key <YOUR_API_KEY_ID> --include-value
```

5.2. Limitación de Tasa (Throttling) en el Usage Plan

Para proteger la API frente a abusos y mantener la calidad del servicio, se ha configurado un mecanismo de *throttling* en el `UsagePlan` asociado a la `API Key`.

La configuración limita el tráfico a un máximo de **5 peticiones por segundo** (`RateLimit = 5`) con un **burst** máximo de **10** (`BurstLimit = 10`).

El parámetro `RateLimit` define el ritmo sostenido de solicitudes, mientras que el `BurstLimit` permite manejar picos breves de hasta 10 peticiones simultáneas antes de aplicar la limitación. En otras palabras, se toleran ráfagas cortas sin penalizar inmediatamente al cliente, pero si el ritmo supera el límite durante más de unos segundos, las peticiones excedentes serán bloqueadas o retrasadas.

5.3. Seguridad de Red e Infraestructura

Además de la clave de API, la infraestructura está diseñada para no ser accesible desde el exterior.

- **Aislamiento de la Base de Datos (DynamoDB):** DynamoDB es un servicio regional gestionado por AWS, inaccesible directamente desde Internet. Su acceso se realiza solo a través de la API de AWS y está estrictamente controlado por roles de IAM. En nuestra configuración, las Lambdas y ECS asumen el rol (`LabRole`) que les da permisos explícitos para interactuar con DynamoDB, garantizando el acceso solo a nuestro cómputo autorizado.
- **Aislamiento del Cómputo (Arquitectura Acoplada ECS):** En la plantilla `acoplada.yml`, el contenedor de Fargate está totalmente aislado de Internet gracias a su Grupo de Seguridad (`ECSSecurityGroup`).
    1. Este grupo de seguridad solo permite tráfico entrante (`SecurityGroupIngress`) en el puerto 8080.
    2. Además, el origen de ese tráfico (`CidrIp`) está permitido únicamente a `!Ref VpcCidr`. `VpcCidr` es el rango de direcciones IP internas de la propia VPC (por ejemplo, `10.0.0.0/16`).
    3. Esto significa que ninguna dirección IP pública (`0.0.0.0/0`) puede iniciar una conexión con el contenedor. El único tráfico que acepta es el que se origina dentro de la VPC, que en este diseño proviene del *Network Load Balancer* (NLB) interno. El acceso público se gestiona exclusivamente a través de API Gateway, que se comunica con el NLB mediante el VPC Link.

6. Documentación automática.

Para cumplir con el requisito de puntuación adicional "La API genera documentación de forma automática", ambas plantillas de arquitectura (`acoplada.yml` y `desacoplada.yml`) incluyen una sección de recursos `AWS::ApiGateway::DocumentationPart`.

Estos recursos definen la documentación de la API siguiendo la especificación OpenAPI, detallando:

- **Información General:** Metadatos de la API (Type: API), como la descripción y versión.
- **Recursos:** Descripción de cada ruta (ej. `/tasks` y `/task/{id}`).
- **Métodos:** Detalles específicos para cada operación (ej. `GET /tasks`), incluyendo resúmenes (summary), parámetros (parameters), cuerpos de solicitud (requestBody) y los códigos de respuesta esperados (responses).

Finalmente, se crea un recurso `AWS::ApiGateway::DocumentationVersion` para publicar esta documentación. Una vez desplegada la pila, esta documentación puede ser exportada desde la consola de API Gateway en formato OpenAPI (JSON o YAML) para su uso en herramientas de visualización como Swagger UI o para importar en Postman.

7. Pruebas y verificaciones

Para validar el correcto funcionamiento de los cinco endpoints de la API en ambas arquitecturas, se han preparado dos métodos de prueba que cumplen con los requisitos de la práctica.

7.1. Método 1: Batería de Pruebas (Postman)

Se ha creado una colección de Postman (.json) que contiene una batería de pruebas automatizadas para cada uno de los endpoints. Esta colección permite una validación rápida y programática del comportamiento de la API.

El archivo .json estará disponible en el repositorio del proyecto.

Nota: A continuación se proporciona un enlace a la colección, aunque es posible que dicho enlace no esté disponible de forma permanente.

Colección Postman

7.2. Método 2: Interfaz Gráfica Rudimentaria (HTML)

Para cumplir con el requisito de "preparar alguna interfaz rudimentaria de prueba", se ha desarrollado un único archivo `index.html`. Este archivo utiliza JavaScript (fetch) para consumir los cinco (5) endpoints de la API de forma gráfica desde el navegador.

Instrucciones de uso:

1. **Abrir el archivo:** Simplemente se debe abrir el archivo `index.html` en cualquier navegador web localmente (haciendo doble clic en él, no requiere un servidor web).
2. **Configurar front:** Al entrar a la página se solicita la url proporcionada por la plantilla y el apikey para poder hacer las solicitudes.
3. **Probar:** La interfaz permite introducir los datos necesarios (como el ID para buscar o el título/descipción para crear/actualizar) y probar cada una de las funciones CRUD.
4. **Configuración CORS:** Esta prueba local es posible gracias a que, en ambas plantillas de API Gateway (`acoplada.yml` y `desacoplada.yml`), se han

configurado métodos `OPTIONS` que gestionan el pre-flight de CORS. Específicamente, se devuelve la cabecera `Access-Control-Allow-Origin: "*"`, permitiendo que un cliente (como nuestro `index.html` local) realice peticiones desde cualquier origen.

8. Análisis de costes

En este apartado se justifica el pricing de la solución con un tráfico esperado de **1.000.000 de peticiones/mes** en la región `US East (N. Virginia)` y **sin** aplicar la capa gratuita. Se han generado dos estimaciones con la herramienta oficial de AWS (https://calculator.aws/#/) y se **adjuntan dos PDFs** con el detalle de cada arquitectura (`estimacion_acoplada.pdf` y `estimacion_desacoplada.pdf`).

El tráfico se ha modelado como:

- 600.000 operaciones de lectura (420k Get Task + 180k Get Tasks)
- 400.000 operaciones de escritura (200k Post Task + 100k Put Task + 100k Delete Task)

8.1. Arquitectura acoplada (ECS Fargate)

Coste estimado: $44.61/mes (**$535.32/año**).
Enlace a calculadora (Disponible 1 año): Estimación Acoplada.

Servicio	Configuración (alto nivel)	Coste/mes
API Gateway (REST)	1M peticiones/mes, sin caché	$3.50
ECS Fargate	1 tarea, 730 h/mes, almacenamiento efímero 20 GB	$9.01
Network Load Balancer (NLB)	1 NLB interno, conexiones TCP	$16.44
AWS PrivateLink (VPC Link)	1 conexión para API GW ↔ NLB	$14.61
CloudWatch Logs	0.5 GB ingerido + 0.5 GB entregado	$0.50
DynamoDB (On-demand)	1 GB de datos, items <1 KB	$0.54
ECR	Imagen ∼76 MB almacenada	$0.01
Total		**$44.61**

Cuadro 1: Resumen de configuración y costes – arquitectura acoplada.

8.2. Arquitectura desacoplada (Serverless Lambda)

Coste estimado: $5.01/mes (**$60.12/año**).
Enlace a calculadora (Disponible 1 año): Estimación Desacoplada.

Servicio	Configuración (alto nivel)	Coste/mes
API Gateway (REST)	1M peticiones/mes, sin caché	$3.50
AWS Lambda (Get by ID)	420k invoc./mes, duración media 60 ms	$0.18
AWS Lambda (Get tasks)	180k invoc./mes, duración media 80 ms	$0.10
AWS Lambda (Post task)	200k invoc./mes, duración media 60 ms	$0.09
AWS Lambda (Put task)	100k invoc./mes, duración media 60 ms	$0.05
AWS Lambda (Delete task)	100k invoc./mes, duración media 40 ms	$0.04
CloudWatch Logs	0.5 GB ingerido + 0.5 GB entregado	$0.50
DynamoDB (On-demand)	1 GB de datos, items <1 KB	$0.54
ECR	Imagen ∼147 MB almacenada	$0.01
Total		**$5.01**

Cuadro 2: Resumen de configuración y costes – arquitectura desacoplada.

Anotaciones

- La **arquitectura acoplada** mantiene unos costes fijos significativos asociados al uso de *Network Load Balancer* y *PrivateLink*, los cuales constituyen los principales componentes del gasto mensual.
- En la **arquitectura desacoplada**, el mayor impacto en costes proviene del servicio *API Gateway*, mientras que el resto de componentes mantienen un precio minúsculo.

9. Comparativa y punto de equilibrio

9.1. Resumen comparativo entre arquitecturas

Ambas arquitecturas propuestas ofrecen un cumplimiento pleno de los requisitos definidos en la práctica. Sin embargo, presentan diferencias importantes en su diseño, comportamiento y costes.

En la **arquitectura acoplada**, todo el servicio de la API se ejecuta dentro de un contenedor monolítico desplegado sobre ECS Fargate. Esto implica que el coste principal se concentra en los recursos fijos asociados al servicio de cómputo y a la red (Fargate, NLB y VPC Link) independientemente del volumen de peticiones procesadas. La latencia es estable una vez que la tarea está en ejecución, y el entorno ofrece un control total del runtime, pero también exige una mayor carga operativa en cuanto al mantenimiento.

Por otro lado, la **arquitectura desacoplada** se basa en una filosofía completamente serverless. Cada endpoint de la API es atendido por una función Lambda independiente, que se ejecuta para procesar la petición. Este modelo elimina los costes fijos de cómputo, sustituyéndolos por un esquema de facturación variable según el número de invocaciones y su duración. A cambio, la arquitectura presenta una latencia algo superior en los arranques en frío (*cold starts*) y una complejidad mayor. En términos operativos la carga de mantenimiento se reduce drásticamente, dado que AWS gestiona toda la infraestructura subyacente.

En resumen, la arquitectura acoplada ofrece un control más fino sobre el entorno de ejecución y una latencia predecible, mientras que la arquitectura desacoplada prioriza la eficiencia económica y la escalabilidad automática, resultando especialmente adecuada para cargas variables o impredecibles. A partir de cierto volumen de tráfico, no obstante, el modelo serverless puede igualar o incluso superar el coste del despliegue tradicional, como se analiza a continuación.

9.2. Predicción: punto de equilibrio de costes entre arquitecturas

Partiendo de los supuestos del apartado 8 (volumen de tráfico, tiempos de ejecución, memoria y región de despliegue), el coste mensual estimado de cada enfoque puede expresarse como:

$$C_{\text{acoplada}}(N) \approx C_{\text{API GW}}(N) + C_{\text{DDB}}(N) + C_{\text{infraestructura\_ECS}}$$
$$C_{\text{desacoplada}}(N) \approx C_{\text{API GW}}(N) + C_{\text{DDB}}(N) + C_{\lambda}(N)$$

Como los componentes de API Gateway y DynamoDB impactan por igual en ambas alternativas, el **punto de equilibrio** se alcanza cuando:

$$C_{\text{infraestructura\_ECS}} \approx C_{\lambda}(N)$$

Parámetros considerados:

- **Coste base de la arquitectura acoplada:** ECS \$9.01 + NLB \$16.44 + VPC Link \$14.61, resultando en un total aproximado de **\$40.1/mes**.
- **Coste Lambda promedio (mezcla CRUD y duraciones estimadas):** Suma ponderada de ejecuciones: \$0.19 + \$0.11 + \$0.09 + \$0.04 + \$0.03 = **\$0.46** por cada **1 millón de invocaciones**.
  Esto equivale a un coste de aproximadamente $\mathbf{\$4.6 \times 10^{-7}}$ por petición.

Estimación del umbral:

$$N_{\text{equilibrio}} \approx \frac{40{,}1}{4{,}6 \times 10^{-7}} \approx \mathbf{8{,}7 \times 10^{7}} \text{ peticiones/mes}$$

Conclusión: con las hipótesis anteriores, la arquitectura **desacoplada** empieza a resultar más costosa que la **acoplada** a partir de unas **87 millones** de solicitudes mensuales. Para volúmenes menores la opción *serverless* mantiene una clara ventaja económica.

Consideraciones:

- Si la **duración media** de las Lambdas aumentara, el coste por petición subiría y el umbral bajaría proporcionalmente.
- Si ECS necesitase **DesiredCount >1** por alta demanda, $C_{\text{fijos\_ECS}}$ crecería y el umbral se desplazaría aún más arriba.